The Mathematics of Computing*

Don Johnson

Abstract

An introduction to the mathematics used by computers. Topics include, Base 2, a very short overview of logical operators and how they relate to binary addition, and information on bit representations of real numbers.

We need to concentrate on how computers represent numbers and perform arithmetic. Suppose we have twenty-six "things." This number is an abstract quantity that we need to represent so that we can perform calculations on it. We represent this number with **positional notation** using base 10: $26_{10} = 2 \cdot 10^1 + 6 \cdot 10^0$. We could also use base-8 notation ($32_8 = 3 \cdot 8^1 + 2 \cdot 8^0$), base 16 ($1A = 1 \cdot 16^1 + A \cdot 16^0$, with A representing ten), or base 2 ($11010_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$). Each of these represent the **same** quantity; all numbers can be so represented and arithmetic can be performed in each system. Which we chose is a matter of convenience and cultural heritage. Each position is occupied by an integer between 0 and the base minus 1, and the integer's position implicitly represents how many times the base raised to an exponent is needed. The integer N is succinctly expressed in base-b as $N_b = d_n d_{n-1} \ldots d_0$, which mathematically means $N_b = d_n b^n + \cdots + d_0 b^0$. No matter what base you might choose, addition, subtraction, multiplication, and division all follow the same rules you learned as a child. These rules can be derived mathematically from the positional notation conventions defined by this formula. (Non-positional systems are very strange; take Roman numerals for example. Try adding (or, more extremely, dividing) two numbers using this number representation system). To extend the positional representation to signed integers, we add a symbol that represents whether the number is positive or negative.

Humans have ten fingers, and so it's not so surprising that many cultures throughout history have used base 10. Digital computers use base 2 or **binary** number representation, each digit of which is known as a **bit** (**b**inary dig **it**). Here, each bit is represented as a voltage that is either "high" or "low," thereby representing "1" or "0", respectively. To represent signed values, we tack on a special bit – the **sign bit** – to express the sign. The binary addition and multiplication tables are

$$0+0=0, 1+1=10, 0+1=1, 1+0=1, 0 \cdot 0=0, 0 \cdot 1=0, 1 \cdot 1=1, 1 \cdot 0=0. \quad (1)$$

A **carry** means that a computation performed at a given position affects other positions as well. Here, $1+1=10$ is an example of a computation that involves a carry. Note that if carries are ignored, subtraction of two single-digit binary numbers yields the same bit as addition. Computers use high and low voltage values to express a bit, and an array of such voltages express numbers akin to positional notation. Logic circuits perform arithmetic operations.

*Version 2.3: Aug 10, 2004 1:20 pm GMT-5
†http://creativecommons.org/licenses/by/1.0

Exercise 1 *(Solution on p. 4.)*

Add twenty-five and seven in base 2. Note the carries that might occur. Why is the result "nice"?

Note that the logical operations of AND and OR are equivalent to binary addition (if carries are ignored). When used, logical operators indicate truth or falsehood. For example, the statement $A \wedge B$, which represents "A AND B", will be true if **both** A is true and B is true, and false otherwise. You could use this kind of statement to tell a search engine that you want to restrict hits to cases where A and B occur together, and not other cases. The statement $A \vee B$ represents "A OR B". It will be true if A is true, or B is true, or both, and false if A and B are both false. Note that if we represent truth by a "1" and falsehood by a "0", binary multiplication corresponds to AND statements, and binary addition corresponds to OR. The Irish mathematician George Boole discovered this equivalence in the mid-nineteenth century. It laid the foundation for what we now call Boolean algebra, which expresses logical statements as equations. More importantly, any computer using base-2 representations and arithmetic can also easily evaluate logical statements. This fact makes an integer-based computational device much more powerful than might be immediately obvious.

Computers express numbers in a fixed-size collection of bits, commonly known as the computer's **word length**. Today, word-lengths are either 32 or 64 bits, corresponding to a power-of-two number of bytes (8-bit "chunks"). This design choice restricts the largest integer (in magnitude) that can be represented on a computer.

Exercise 2 *(Solution on p. 4.)*

For both 32-bit and 64-bit integer representations, what is the largest number that can be represented? Don't forget that the sign bit must also be included.

When it comes to expressing fractions, positional notation is easily extended to negative exponents using the decimal point convention: Wherever the decimal point occurs in a string of digits, we know that the first digit to the left corresponds to the zero exponent, and the one to the right is -1 . In this way, we have $2.5_{10} = 2 \times \times 10^0 + 5 \times \times 10^1$. Computers **don't** use this convention to represent numbers with fractional parts; instead, they use a generalization of scientific notation. Here, a number (not necessarily an integer) x is expressed as $m \times b^e$, with the **mantissa** m lying within a proscribed range (scientific notation requires $1 \leq |m| < 10$), b is the base, and e is the **integer exponent**. Computers use a convention known as **floating point**, where base 2 is used and the mantissa must lie between one-half and one. Thus, two and one-half in the computer's number representation scheme becomes five-eighths times four: $.101 \times \times 2^2$. The mantissa is stored in most of the bits in a computer's word, and the exponent stored in the remaining bits. Both the exponent and mantissa have sign bits associated with them. Thirty-two-bit floating point numbers, known as single-precision floating point, have eight bits representing the exponent (including the sign bit) and the remaining twenty-four representing the mantissa (again, also including a sign bit).

Exercise 3 *(Solution on p. 4.)*

What are the largest and smallest numbers that can be represented in 32-bit floating point? 64-bit floating point that has sixteen bits allocated to the exponent? Note that both exponent and mantissa require a sign bit.

So long as the integers aren't too large, they can be represented exactly in a computer using the binary positional notation. Electronic circuits that make up the physical computer can add and subtract integers without error. However, this statement isn't quite true; when does addition cause problems? Floating point representation handles numbers with fractional parts, but only some with no error. Similar to the integer case, the number could be too big or too small to be so represented. More fundamentally, **many** numbers cannot be accurately represented no matter how many bits are used for the exponent and mantissa. For example, you know that the fraction $\frac{1}{3}$ has an infinite decimal (base 10) expansion; no matter how many decimal digits you have in your calculator, you will never represent $\frac{1}{3}$ with complete accuracy. It also has an infinite binary expansion as well.

Exercise 4 *(Solution on p. 4.)*

Can $\frac{1}{3}$ be expressed in a finite number of digits in any base? If so, what bases do the job?

Clearly, many numbers have infinite expansions, and this situation applies to binary expansions as well. Consequently, number representation and arithmetic performed by a computer cannot be infinitely accurate in most cases. The errors incurred in most calculations will be small, but this fundamental source of error can cause trouble at times.

Solutions to Exercises in this Module

Solution to Exercise (p. 1)
$25_{10} = 11001_2$ and $7_{10} = 111_2$. We find that $11001_2 + 111_2 = 100000_2 = 32_{10}$.
Solution to Exercise (p. 2)
For b -bit signed integers, the largest number is $2^{b-1} - 1$. For $b = 32$, we have 2,147,483,647, and for $b = 64$, we have 9,223,372,036,854,775,807 or about $9.2 \times \times 10^{18}$.
Solution to Exercise (p. 2)
In floating point, the number of bits in the exponent determines the largest and smallest representable numbers. For 32-bit floating point, the largest (smallest) numbers are $2^{\pm 127} = 1.7 \times \times 10^{38} \cdot 5.9 \times \times 10^{39}$. For 64-bit floating point, the largest number is about 10^{9863} .
Solution to Exercise (p. 2)
If the base is a multiple of three, then $\frac{1}{3}$ will have a finite expansion. For example, in base 3: $\frac{1}{3} = 0.1_3$, and in base 6: $\frac{1}{3} = 0.2_6$.